

Scientific Programming Using Java and C: A Remote Sensing Example

Donald Prados, Michael Johnson, Mohamed A. Mohamed, Changyong Cao, Jerry Gasser,
Don Powell, and Lloyd McGregor
Lockheed Martin Space Operations - Stennis Programs
Stennis Space Center, Mississippi 39529

ABSTRACT

This paper presents results of a project to port code for processing remotely sensed data from the UNIX environment to Windows. Factors considered during this process include time schedule, cost, resource availability, reuse of existing code, rapid interface development, ease of integration, and platform independence. The approach selected for this project used both Java and C. By using Java for the graphical user interface and C for the domain model, the strengths of both languages were utilized and the resulting code can easily be ported to other platforms. The advantages of this approach are discussed in this paper.

Key Words: Remote sensing, aircraft, Java, JNI

I. INTRODUCTION

Scientists at NASA's John C. Stennis Space Center (SSC) recently completed a pilot project focused on streamlining the code and algorithms used to process certain types of remotely sensed data. In the course of this project, they developed a method for porting this code, which was written in C, from the UNIX environment to Windows. This software was originally developed to process data sets acquired by the Airborne Terrestrial Applications Sensor (ATLAS) and Calibrated Airborne Multispectral Scanner (CAMS) developed at SSC under NASA's Commercial Remote Sensing Program (CRSP) Office.

ATLAS is a 15-channel (band) airborne multispectral sensor system designed for ground spatial resolution between 2 - 25 meters and spectral coverage between 0.45 - 12.2 μm . ATLAS has 9 reflective bands and 6 thermal bands. The ninth reflective band, however, is unusable because of extremely low signal-to-noise ratio (Cao, 1999).

CAMS is a 9-channel airborne multispectral sensor system designed for ground spatial resolution between 2 - 25 meters and spectral coverage between 0.45 - 12.5 μm . The first 8 bands are reflective bands, and band 9 is a thermal band.

ATLAS and CAMS data files are flat files composed of a series of flight lines. Each flight line is organized as a collection of scan lines. Scan lines are composed of a header block followed by the video portion of the image data. The header block contains all the housekeeping data, including flight parameters, ambient conditions, and calibration data. The video data block consists of n channels of video data. The maximum value of n is 15 for ATLAS and 9 for CAMS. The exact channel numbers that were recorded on each flight mission are stored in the header block.

The ATLAS and CAMS data processing software was originally developed in C, and the user interface was developed using the ERDAS IMAGINE Top! Kit. ERDAS IMAGINE is a commercial off the shelf (COTS) software package used to process remote sensing imagery. As the software evolved, the labor required to manage and maintain the code increased significantly. The system was also limited to running on UNIX workstations and required modifications whenever an updated version of ERDAS IMAGINE was received. Additionally, data sets created by the software were in IMAGINE's proprietary data format.

To alleviate this growing burden, the CRSP Verification and Validation (V&V) team (Mohamed, 1997) initiated a pilot project focused on streamlining the code and algorithms used to process ATLAS and CAMS data. The primary objective of this pilot project was to develop an algorithm validation process and to further identify and address the

weaknesses of the validation process, allowing it to serve as a refined model for others to use. The second objective was to comply with ISO-9000 standards concerning documentation, software development, and certification of data processing activities. Additional objectives were to reduce processing time, to gain platform independence, and to remove the restrictions of a proprietary data format.

As part of the V&V effort, the code was simplified and streamlined in the process of making it ISO-9000 compliant and more maintainable. To gain platform independence, a new, object-oriented user interface was written in the Java programming language. The proprietary data format was replaced by a conventional format that allows output images to be imported into other image processing applications. By removing the dependence on proprietary software, the application can easily be distributed with the data to the users. Several enhancements were also added to improve the user interface and to speed processing of the data sets.

Code for breaking up a raw ATLAS or CAMS image file into separate housekeeping data and image data files was ported, along with an algorithm for performing radiometric calibration. The porting process included removing platform-dependent code, writing C functions according to the Java Native Interface (JNI) convention (Sun Microsystems, 1999), and compiling the C code into dynamic link libraries (DLL's). The DLL's are loaded and executed from Java using JNI. This allows legacy remote sensing code that has already been verified and validated on the UNIX platform to be reused with few changes on other platforms. Section III provides an example of JNI.

New code was written in C for automatically identifying and separating flight lines and for performing additional radiometric calibration. The new code was written in C for improved performance and for ease of use, because the programmers have greater expertise in C than in Java. As with the ported code, the new code was written in a platform-independent manner and compiled into DLL's to be called from Java using JNI. Since the Java code is platform independent, porting the code to another platform requires little more than recompiling the C libraries on the new platform.

Two core Java features greatly simplified the ability to display ATLAS and CAMS housekeeping and image data. First, the Image class and related classes made the creation and the display of images relatively easy. Second, the JTable class and related classes made the creation and the display of housekeeping data much easier.

II. ADPA OVERVIEW

The Aircraft Data Processing Algorithms (ADPA) software (Lockheed Martin Stennis Operations, 1998) allows the user to read an ATLAS or CAMS image file from tape, split the image file into separate flight line files, split a flight line file into separate housekeeping data and image data files, and radiometrically calibrate the image data. The user can view the image file before and after splitting it into flight lines, can view image data as a grayscale or an RGB image, and can view the housekeeping data in a scrollable table. Bad housekeeping data and bad image data are automatically flagged. The user can mark, fix, and archive both the bad housekeeping data and the bad image data.

Read ATLAS and CAMS Data from Tape

The first step in processing an ATLAS or a CAMS image is to read the raw image from tape, creating a single file on the hard disk. The data dump (dd) utility is used to read the image from tape. Versions of the dd utility exist for both UNIX and Windows platforms. This newly created file contains both housekeeping data and image data for all flight lines.

The extracted file can be viewed by ADPA using the Java Image class and associated classes. Since this file is often several hundred megabytes in size, the user is given the option of selecting only the band or bands of interest, a beginning line within the image, and a line count. Viewing the raw image allows the user to visually identify the flight lines in the image before splitting the image into separate flight lines.

Split Image into Individual Flight Lines

The next step automatically splits the image file read from tape into several files: one for each flight line. The code for splitting flight lines was written in C and compiled into a DLL. The user selects the file to split through the Java

graphical user interface (GUI), and the Java code calls the DLL through JNI. Each raw file created contains both housekeeping data and image data for a single flight line. The user can select the minimum number of scan lines required for a valid flight line.

The created files can also be viewed by ADPA after the user selects the instrument type (ATLAS or CAMS), the band or bands of interest, a beginning line, and a line count. As with the raw image read from tape, the user is shown both a table of the housekeeping information and an image from the selected bands. ADPA also performs quick checks for bad housekeeping and bad image data, displaying any bad data found in tables.

Split a Flight Line into Separate Housekeeping Data and Image Data Files

The C code for splitting an image file into separate housekeeping data and image data files was ported from UNIX and compiled into a DLL. After the user selects an ATLAS or CAMS image file containing a single flight line, the bands of interest, a beginning line, and a line count, the DLL is called from Java using JNI.

A header file associated with the flight line is created that contains metadata information, including the instrument type, the pixel size, and the bands selected. The pixel size can be 8 bits (byte), 16 bits (short integer), or 32 bits (floating point). The header file also keeps track of other files created for this flight line, including output from radiometric processing.

Radiometric Processing

At present ADPA implements two separate algorithms, TRADE (Spiering, 1994) and Radiometry, for performing radiometric calibration on ATLAS and CAMS images. The TRADE algorithm creates an image data file containing calibrated thermal bands and unchanged reflective bands. The Radiometry algorithm creates an image data file containing both calibrated reflective bands and calibrated thermal bands. These algorithms were written in C and compiled into DLL's, and they are called from Java using JNI. The TRADE code was ported from UNIX, and the Radiometry code was developed as an enhancement to the new system.

The TRADE algorithm uses an estimated system transfer equation and an atmospheric model generated by LOWTRAN. TRADE converts thermal image digital numbers to radiance units, adjusts the radiance values for atmospheric effects, and, optionally, estimates ground temperature using the radiance values. The algorithm takes as input a calibration file from the sensor Performance Verification Test and a LOWTRAN file containing the atmospheric model generated by LOWTRAN 6 or LOWTRAN 7.

The Radiometry algorithm performs radiometric calibration on the reflective bands as well as the thermal bands, using blackbody values (low and high), a spectral response curve, data from the onboard integrating sphere, and calibration lamp values. Figure 1 shows the input screen for the Radiometry algorithm. The user can select the type of output (in-band or average spectral radiance), the output pixel size, an emissivity file, a spectral response file, an integrating sphere file, and a cutoff frequency for use by a low-pass filter.

Data Type Selection	ATLAS
Output Radiance Type Selection	IN-BAND
Output Format Type	Float
Input File	mx9805f1.all.t2
Emissivity File	atlas.ems
Spectral Response File	atlas798.src
Integrating Sphere File	atlas.isr
Cutoff Frequency (0.0 to 5.0 hz)	0.5
Reflective Band Number For Graph	5

Process Cancel View Graphs

Figure 1. Input Screen for Radiometric Calibration Using the Radiometry Algorithm

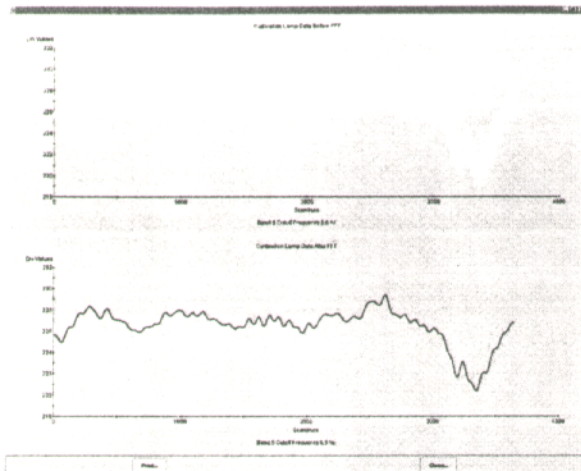


Figure 2. Calibration Lamp Data Before and After Smoothing

Because of noise in the low blackbody and calibration lamp data needed for calibration, a low-pass filter is used to remove high-frequency components from the calibration data.

Graphs of these filtered housekeeping values can be displayed, showing values before and after smoothing. Figure 2 shows the graphs for calibration lamp data. This chart was created using the JCCart class (KL Group, 1999). Numerous controls and components, such as JCCart, are distributed with most Java development environments. Creating charts using this component was much easier than creating charts from scratch.

The Radiometry input screen also allows the user to choose between in-band or average spectral radiance. Average spectral radiance is obtained by dividing the in-band radiance by the integral of the spectral response for a particular band. In-band radiance is useful for examining the total radiance for a particular band, while the average spectral radiance is used for cross band studies.

View Image Data as RGB Image

After splitting an ATLAS or CAMS flight line into separate housekeeping data and image data files, the user can open the image data file and display an image in a scrollable window (see Figure 3). Images from radiometrically calibrated image data files can also be displayed. The user selects the image data file to view, the pixel size, the bands of interest, a beginning line, and a line count. If an associated header file exists for the selected image data file, the instrument type, pixel size, and bands selected will be read from the header file.

If less than three bands are selected, a gray-level image is displayed; otherwise, an RGB image is displayed using three of the selected bands. Input fields allow the user to change the bands used to create the image. Since all bands selected are read into random access memory, changing the bands to be used for red, green, and blue does not require additional disk access, resulting in a relatively quick redraw of the image using the new bands. The Java Image class and associated classes make creating and displaying an image from the selected bands relatively simple.

Crosshairs, which can be turned off or on, show the location and the value at the location for all bands in the image. Menu selections allow the user to mark a line as bad, mark a line as good that was previously marked as bad, and select the starting column and ending column of the bad part of the line. Another menu selection allows the user to display the image data bad line table.

When opening the image data file, ADPA performs a quick check for bad image data, displaying any bad lines found in a bad line table. Currently, a line in an image is flagged as bad if one of the calibration data values for that line is saturated. Figure 4 shows the bad line table for the image data of Figure 3. Input fields allow the user to select the starting column and ending column of the bad part of the line and to fix the bad part of the line. It can be fixed by averaging the line above and the line below, by copying the line above, by copying the line below, or by shifting



Figure 3. ATLAS RGB Image Using Bands 4, 2, and 1 for Red, Green, and Blue

the selected portion of the line by one or more columns. Lines can be marked as fixed using a checkbox. The user can also undo the last change and can save or print the table.

View Housekeeping Data in Scrollable Table

After splitting an ATLAS or CAMS flight line into separate housekeeping data and image data files, the user can open and view the contents of the housekeeping file. As usual, the user can first select a beginning line and a line count to view only part of the table.

Figure 5 shows a housekeeping table with two lines marked as bad. Menu selections allow the user to mark a line as bad, mark a line as good, and select the starting column and ending column of the bad part of the line. Another menu selection allows the user to display the housekeeping bad line table.

Figure 5. ATLAS Housekeeping Table Showing Two Bad Lines

Experience has shown that ATLAS and CAMS housekeeping data often have some bytes missing with the following bytes shifted to the left. If this is the problem with the bad line, the preferred approach is to fix the line by shifting to the right. The user would select the beginning and ending column of the part of the line to shift, shift it to the right, select the beginning and ending column of the part of the line with missing data, and fix it by averaging the line above and the line below. Most columns in the housekeeping table can also be directly edited if averaging does not make sense.

III. JAVA/C INTERFACES

C code ported from UNIX and compiled into DLL's can easily be called from Java using JNI. JNI allows legacy remote sensing code that has already been verified and validated on the UNIX platform to be reused with few

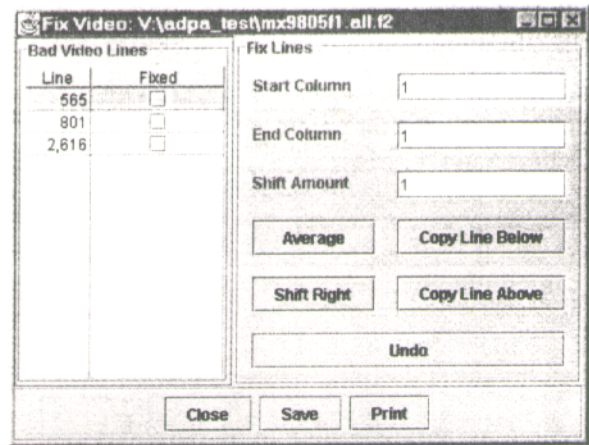


Figure 4. Bad Lines Found in Image Data

Both the housekeeping table and the bad line table use the Java JTable class and related classes. The JTable class has many facilities that allow custom rendering and editing but provides defaults for these features so that simple tables can be set up easily.

Figure 6 shows the bad line table for the housekeeping data shown in Figure 5. Input fields allow the user to select the starting column and ending column of the bad part of the line and to fix the bad part of the line. It can be fixed by averaging the line above and the line below, by copying the line above, by copying the line below, or by shifting the selected portion of the line by one or more columns. The user can also undo the last change and can save or print the table.

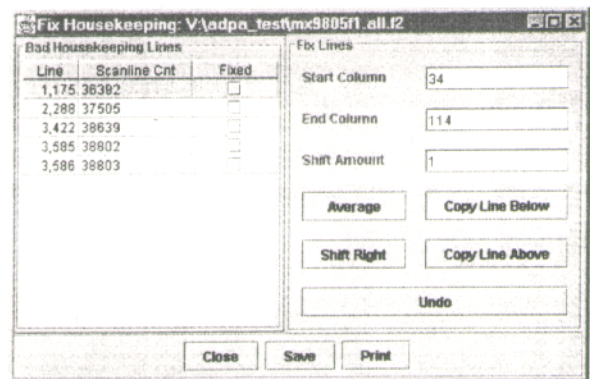


Figure 6. Screen for Fixing Bad Lines in Housekeeping Data

changes. The programmers removed platform dependent code, including the user interface code, and wrote C functions, according to JNI conventions, that could be called from Java.

ADPA is based upon the Model-View-Controller (MVC) architecture and uses Java threads extensively. Put simply, Model objects contain data, View objects display data, and Controller objects perform actions in response to events. For example, when the user selects Radiometry from the Process menu, the Radiometry input screen, a View object, is displayed (Figure 1). Selecting the Process button results in executing the runRadiometry() method of the Controller object, AircraftController. This method starts the RadiometryThread thread, passing in a reference to the Controller object. This thread has access to the data entered in the input screen, contained in a Model object, through the reference to the Controller object. Figure 7 shows the Java code for interfacing with the C code. All of the methods of the AircraftController class except runRadiometry() were removed for brevity.

```
// The Controller in the MVC architecture
public class AircraftController
{
    // One of many methods in the Java
    // AircraftController class
    public void runRadiometry()
    {
        // Create and start the thread
        new RadiometryThread(this).start();
    }
}

// Radiometry Thread class
public class RadiometryThread extends Thread
{
    // A reference to the controller
    AircraftController controller = null;

    public RadiometryThread(AircraftController cont)
    {
        controller = cont;
    }

    // Executes when the start() method of the Thread
    // class is called.
    public void run()
    {
        Radiometry radiometry = new Radiometry();
        if(!radiometry.isLoaded())
            return;

        int ret = radiometry.runRadiometry(controller);
    }
}

// class to load the dll and execute one of its functions
public class Radiometry
{
    static boolean loaded = false;

    public Radiometry() {}

    static
    {
        try
        {
            // Load Radiometry.dll
            System.loadLibrary("Radiometry");
            loaded = true;
        }
        catch(Exception ex)
        {
            System.out.println(ex.getMessage());
            System.out.println
            ("loadLibrary(\"Radiometry\") unsuccessful");
        }
    }

    public static boolean isLoaded()
    {
        return loaded;
    }

    // Run Radiometry.dll
    public native int
    runRadiometry(AircraftController cont);
}
```

Figure 7. Java Code that Loads Radiometry.dll and Executes One of its Functions

The runRadiometry() method of the AircraftController class creates a RadiometryThread object, passing a reference to the AircraftController, and starts the thread. The RadiometryThread class creates an object of the Radiometry class, loads the associated DLL if it is not loaded, and calls the native method, runRadiometry(), of the Radiometry class. A native method is a platform-specific method that is implemented in a programming language other than Java.

The first time an instance of the Radiometry class is created, the static variable, *loaded*, is initialized to *false*, and the static block is executed. In the static block, Java attempts to load Radiometry.dll. If loading is successful, *loaded* gets set to *true*.

Figure 8 shows the header file, Radiometry.h, and part of one of the C files, Radiometry.c, that make up Radiometry.dll. When the runRadiometry() method is called from within RadiometryThread, the C function in Radiometry.c, Java_Radiometry_runRadiometry(), is called. The name of this method takes the form *Java_classname_functionname*. Running the javah.exe utility, which is part of the Java Development Kit (jdk), on the Java file Radiometry.java automatically generates the header file.

Once the C header file is generated, the user can copy the contained function prototype to a C file and implement the function. Radiometry.c shows the implementation of Java_Radiometry_runRadiometry(). By passing this function a reference to the Java AircraftController object, the C function has access to all of the public methods of the Java AircraftController class.

Since AircraftController has access to the data entered in the Radiometry input screen, methods of AircraftController can be called to retrieve this information. To call these methods, the C code needs the AircraftController object, passed in as an argument to Java_Radiometry_runRadiometry(), and the AircraftController class, obtained from the AircraftController object using the GetObjectClass() method of the JNIEnv class.

After obtaining references to the AircraftController object and the AircraftController class, the radiometry() function is called to execute the C code for performing radiometric calibration. The showMessageDialog() function shows how the user can call a Java method from C. In this case a Java message dialog is displayed, indicating completion of the task.

```

// Radiometry.h generated using javah.exe from jdk

/* DO NOT EDIT THIS FILE - it is machine
generated */
#include <jni.h>
/* Header for class Radiometry */

#ifdef _Included_Radiometry
#define _Included_Radiometry

#ifdef __cplusplus
extern "C" {
#endif

/*
 * Class:      Radiometry
 * Method:     runRadiometry
 * Signature:  (LAircraftController;)I
 */
JNIEXPORT jint JNICALL
Java_Radiometry_runRadiometry
(JNIEnv *, jobject, jobject);

#ifdef __cplusplus
}
#endif

#endif

// Radiometry.c

#include "Radiometry.h"

// Pointer to JNI environment class
JNIEnv *env;

// The AircraftController object
jobject controller;

// The AircraftController class
jclass AircraftController;

JNIEXPORT jint JNICALL
Java_Radiometry_runRadiometry(JNIEnv *_env,
jobject obj, jobject cont)
{
    // Return value
    int ret;

    // Get the JNI environment
    env = _env;

    // Get the AircraftController object
    controller = cont;

    // Get the AircraftController class for the
    // AircraftController object
    AircraftController =
        (*env)->GetObjectClass(env, controller);

    // call radiometry
    ret = radiometry();
    return ret;
}

int radiometry()
{
    int rc;

    // This function does the calibration
    rc = calibrate_main();

    // C function to popup Java message dialog
    showAlertDialog("Done", "Radiometry",
        INFORMATION_MESSAGE);

    return rc;
}

// The C function to call a Java method
void showAlertDialog(char* message,
    char* title, int type)
{
    // Get the AircraftController method
    jmethodID id = (*env)->GetMethodID(env,
        AircraftController, "showAlertDialog",
        "(Ljava/lang/String;Ljava/lang/String;I)V");
    // passes (String, String, int); returns void

    // Convert input arguments from char* to jstring
    jstring jmessage = (*env)->NewStringUTF(env,
        message);
    jstring jtitle = (*env)->NewStringUTF(env, title);

    // Call the Java method from C
    (*env)->CallVoidMethod(env, controller, id,
        jmessage, jtitle, type);

    // Free resources created with NewStringUTF
    (*env)->ReleaseStringUTFChars(env, jmessage,
        message);
    (*env)->ReleaseStringUTFChars(env, jtitle, title);
}

```

Figure 8. C Code for Interfacing with Java

IV. CONCLUSIONS

The advantages of object-oriented design and programming were utilized by using Java for the user interface. Although using C++ or another object-oriented programming language could have been used, Java provides better platform independence. The object-oriented approach not only made rapid application development easier, but it also makes modifications and enhancements to the code easier. For example, adding the ability to display CAMS images and CAMS housekeeping required only minor modifications to the code for displaying ATLAS data. Also, modifying the image display code to handle 16 bit (short) and 32 bit (float) pixels was only a minor modification to the code for displaying 8 bit (byte) pixels.

A major advantage of using Java is that several important features are part of the language, including imaging, tables and other GUI components, threads, and JNI. The imaging and table classes make display of images and housekeeping data simple and portable. The Thread class makes it easy to run in separate threads such time-consuming processes as reading and displaying large images or running radiometric calibration algorithms. JNI makes interfacing Java with other languages, such as C and C++, relatively simple.

Without multithreading, Java GUI screens are unresponsive to being moved or minimized while a time-consuming process is underway. With multithreading, screens and menus are still responsive to user input, and a progress bar can provide visual feedback that a time-consuming process is taking place.

Combining Java and C using JNI allows legacy remote sensing code that has already been verified and validated on the UNIX platform to be reused on other platforms with few changes. It also allows computationally intensive code to be written in C while writing portable user interface code in Java.

The platform-independent nature of the application and its independence from COTS software allow for easy distribution of the application with the image data. Furthermore, since the application runs on Windows, it can be installed on a laptop and taken into the field. This feature provides the ability to decide whether a flight should be reflown shortly after landing and before the data are sent to Stennis Space Center for processing.

Approximately four months were spent in the porting process. During this time personnel became familiar with Java and developed new modules and algorithms. Three programmers worked on the software port, spending about 50% of their time on the project. Some of the project time was attributed to test case development and algorithm validation.

One measure of the success of this project is the comments from users regarding how quickly ATLAS and CAMS imagery can now be processed. The ability to process images faster is due in part to the addition of automated features, in particular, the code for splitting flight lines and the code for detecting bad image and housekeeping data.

Based on the results of this project, the authors believe that Java provides a viable solution for the visualization of scientific/engineering data. Although using two languages increases the complexity of the code, the strengths of each language can be utilized. This combination has resulted in a high-performance software application that is platform independent and easy to maintain.

V. REFERENCES

Cao, C., R. Ryan, D. Olive, M. Johnson, M. Mohamed, D. O'Neil, and G. Gasser, *ATLAS At-sensor Radiance Retrieval and Noise Reduction in the In-flight Calibration Data*, Unpublished, 1999.

KL Group, Inc., *JClass Chart*, <http://www.klg.com/jclass/chart/overview.html>, 1999.

Lockheed Martin Stennis Operations, *ADPA System Requirements Specification Document*, Stennis Space Center, SSC-ADPA-001, Rev. A, March 10, 1998.

Mohamed, M. A., *Algorithm Performance Validation*, NASA/CRSP V&V Workshop, 1997.

Spiering, B., *Descriptive Presentation of the ATLAS Channels 10-15, The ELAS Module "TRADE," and Conversion of Digital Numbers to Temperature*, Unpublished, 1994.

Sun Microsystems, Inc., *Java Native Interface*, <http://www.javasoft.com/products/jdk/1.2/docs/guide/jni/index.html>, 1999.

VI. ACKNOWLEDGEMENTS

This work was supported by NASA's Earth Science Enterprise, Commercial Remote Sensing Program Office under contract number NAS 13-650 at the John C. Stennis Space Center, Mississippi.

Export Availability Statement

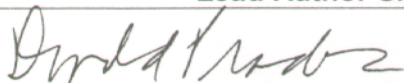
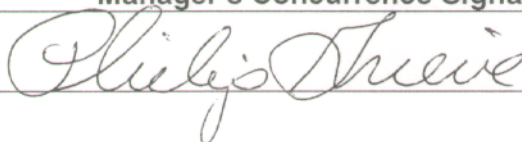
This form must be completed by the lead author, with the assistance of his/her manager, of all CRSP documentation intended for offsite publication and dissemination. Documentation not accompanied by this completed and signed form will not be considered for publication. Help text will appear at the bottom of your screen as you enter each field to be completed. Please complete sections 1-12, print this form, and sign and date the form. Submit the signed form, an electronic copy of your documentation, and a copy of the authors' instructions to Marcia Wise or Rick Lightfoot in Bldg. 1210.

1. Document Title		
Scientific Programming Using Java: A Remote Sensing Example		
2. Journal or Proceedings		
International Symposium on Spectral Sensing Research (ISSSR), Las Vegas, Nevada, October 31-November 4, 1999		
3. Lead Author, Affiliation		
Donald Prados, Lockheed Martin Space Operations - Stennis Programs		
4. Phone Number	5. Fax Number	6. SWR Number for Document Preparation
228-688-1991	228-688-7918	VXD5-2331-20
7. Co-authors, Affiliations		
Mohamed A. Mohamed, Lockheed Martin Space Operations - Stennis Programs Michael Johnson, Lockheed Martin Space Operations - Stennis Programs Changyong Cao, Lockheed Martin Space Operations - Stennis Programs (currently NOAA) Jerry Gasser, Lockheed Martin Space Operations - Stennis Programs Don Powell, Lockheed Martin Space Operations - Stennis Programs Lloyd McGregor, Lockheed Martin Space Operations - Stennis Programs		
8. Security Classification		9. Availability Category
☒ Unclassified	☐ Other	☒ Publicly Available ☐ Other
10. ITAR USML Category Number	11. EAR CCL ECCN Number	
120.11(8)	734.8(c)	
12. Technical Summary of Document Contents as Related to ITAR/EAR Restrictions		
This paper presents results of a project to port code for processing remotely sensed data from the UNIX environment to Windows. By using Java for the graphical user interface and C for the domain model, the strengths of both languages were utilized and the resulting code can easily be ported to other platforms. The advantages of this approach are discussed in this paper. The Java programming language is traditionally used in Internet applications, and the C programming language is traditionally used in scientific applications. Both Java and C are publicly available and widely used. The innovation presented in this paper is the integration of the Java and C languages for scientific programming. This technique could easily be duplicated by anyone with programming experience and is not proprietary. This paper does not discuss any proprietary techniques or methodologies for exploitation of information that might be sensitive to		

intelligence or military applications.

We have determined that the information presented in this paper is not subject to the EAR based on the provisions in Sections 734.3(b)(3)(ii) and 734.8(c) describing fundamental research, or basic and applied research in science and engineering, where the resulting information is ordinarily published and shared broadly within the scientific community, particularly where such research is performed by scientists at a federal agency. Similarly, we have determined that the subject information is not included on the U.S. Munitions List and is exempt from the ITAR because, referring to Section 120.11(8), this paper presents information defined as Public Domain by virtue of its being generally accessible or available to the public through fundamental research in science and engineering at accredited institutions of higher learning in the U.S.

My signature below confirms that I have consulted the International Traffic in Arms Regulations (ITAR) and the Export Administration Regulations (EAR) and that, to the best of my knowledge, the technical content of the document described above meets the export control provisions of the ITAR and the EAR as specified in sections 10 and 11 of this form.

Lead Author Signature	Date
	10-13-99
Manager's Concurrence Signature	Date
	10-19-99

REPORT DOCUMENTATION PAGE					Form Approved OMB No. 0704-0188	
The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.						
1. REPORT DATE (DD-MM-YYYY) 05-10-1999		2. REPORT TYPE		3. DATES COVERED (From - To)		
4. TITLE AND SUBTITLE Scientific Programming Using Java: A Remote Sensing Example				5a. CONTRACT NUMBER NAS13-650		
				5b. GRANT NUMBER		
				5c. PROGRAM ELEMENT NUMBER		
6. AUTHOR(S) Don Prados Lloyd McGregor Mohamed Mohamed Michael Johnson Changyong Cao Jerry Gasser Don Powell				5d. PROJECT NUMBER		
				5e. TASK NUMBER		
				5f. WORK UNIT NUMBER		
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Lockheed Martin Space Operations				8. PERFORMING ORGANIZATION REPORT NUMBER SE-1999-04-00017-SSC		
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Earth Science Application Directorate Commercial Remote Sensing Program Office				10. SPONSORING/MONITOR'S ACRONYM(S)		
				11. SPONSORING/MONITORING REPORT NUMBER		
12. DISTRIBUTION/AVAILABILITY STATEMENT Publicly Available STI per form 1676						
13. SUPPLEMENTARY NOTES Conference - International Symposium on Spectral Sensing Research						
14. ABSTRACT						
15. SUBJECT TERMS						
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON	
a. REPORT	b. ABSTRACT	c. THIS PAGE			Don Prados	
U	U	U	UU	10	19b. TELEPHONE NUMBER (Include area code) (228) 688-1991	